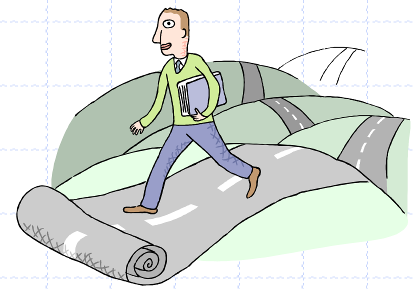
Lesson 13
Weighted Graphs

Wholeness of the Lesson

Weighted graphs are graphs that have *weights* or *costs* associated with each edge. Two questions that often need to be answered when working with weighted graphs are (1) What is the least costly path between two given vertices of the graph? (2) What is the least costly subgraph of the given graph which includes all the vertices of the given graph? Dijkstra's Shortest Path Algorithm provides an efficient solution to the first question; Kruskal's Minimum Spanning Tree Algorithm provides an efficient solution to the second. Solutions to optimization problems of all kinds give expression to Nature's tendency to achieve the most possible with the least expenditure of energy. Waking up to one's own deeper values of intelligence has the effect of drawing Nature's style of functioning into our thinking and action so that we automatically achieve goals with less effort.

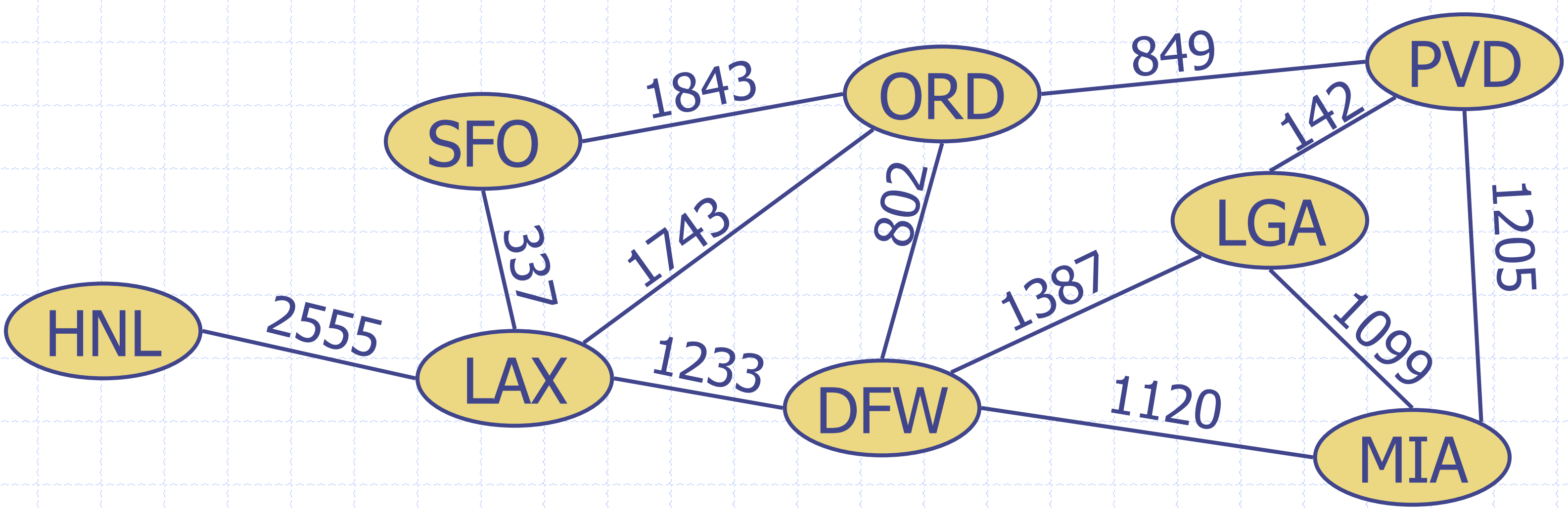
Outline

- Weighted graphs
- Shortest path problem
- Dijkstra's Algorithm
- Minimum spanning tree problem
- Kruskal's Algorithm

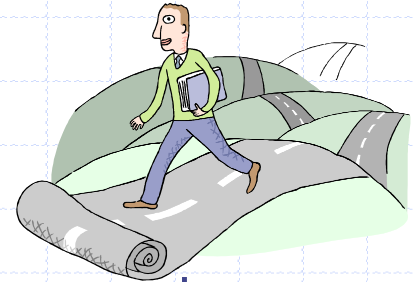


Weighted Graphs

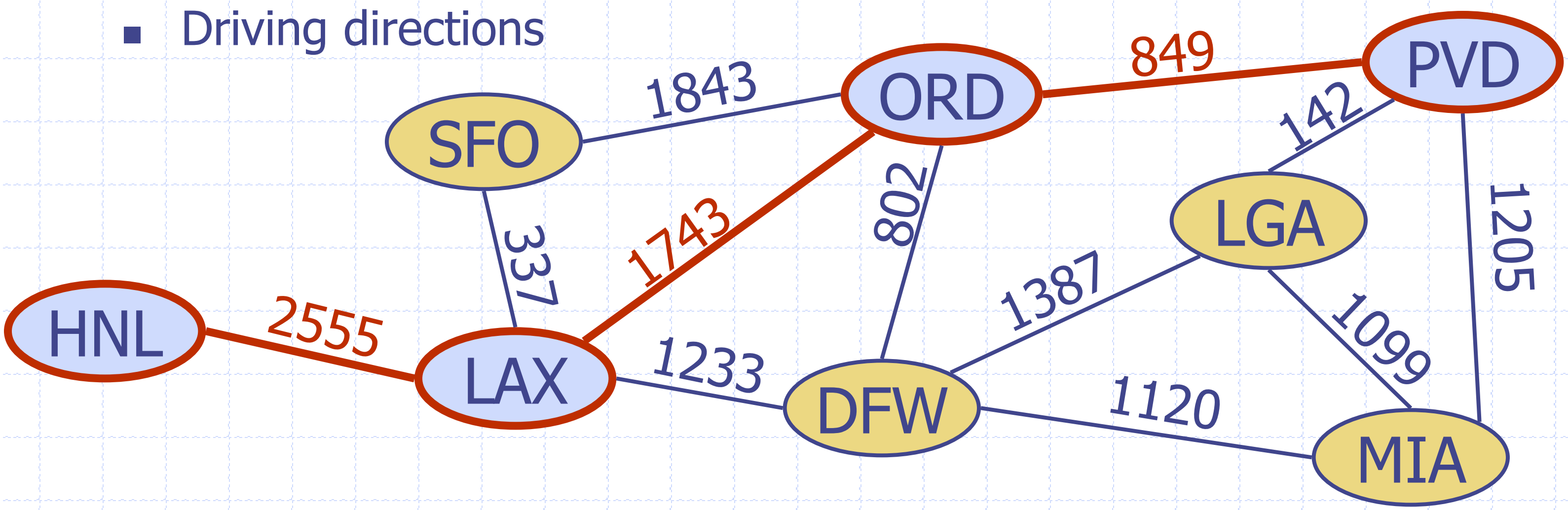
- In a weighted graph, each edge has an associated numerical value, called the weight of the edge (wt: edges → numbers)
- Edge weights may represent distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



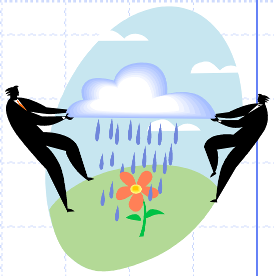
Shortest Path Problem



- Given a connected weighted graph and two vertices s and x, we want to find a path of minimum total weight between s and x.
  - "Length" of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions



Dijkstra’s Algorithm: The Problem



- The *distance* of a vertex $\boldsymbol{v}$ from a vertex $\boldsymbol{s}$, denoted $\boldsymbol{d(s,v)}$, is the length of a shortest path between $\boldsymbol{s}$ and $\boldsymbol{v}$

- ***Question:*** Is it always true in a weighted graph that, for any two vertices v, w, $d(v,w) = wt(v,w)$? Prove or give a counterexample.

- Assumptions:
  - the graph G = (V,E) is connected
  - the edges are undirected
  - the edge weights are nonnegative

- Starting with weighted graph G = (V,E) and starting vertex s, we wish to compute, for each vertex v, the distance from s to v in G.
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex *s*
- We will store our computed value of the distance from s to any vertex v in an array A:

  A[v] = our computed value of distance from s to v
- If our algorithm is right (which we will need to prove) then for each v in V, A[v] = d(s,v).

Dijkstra’s Algorithm

Input: A simple connected undirected weighted graph *G* with nonnegative edge weights, determined by a weight function *wt* (*x,y*), and a starting vertex *s* of *G*.

Output: Array A of distances d(s,v) from s to v, for each v in V, so A[v] = d(s,v) for each v

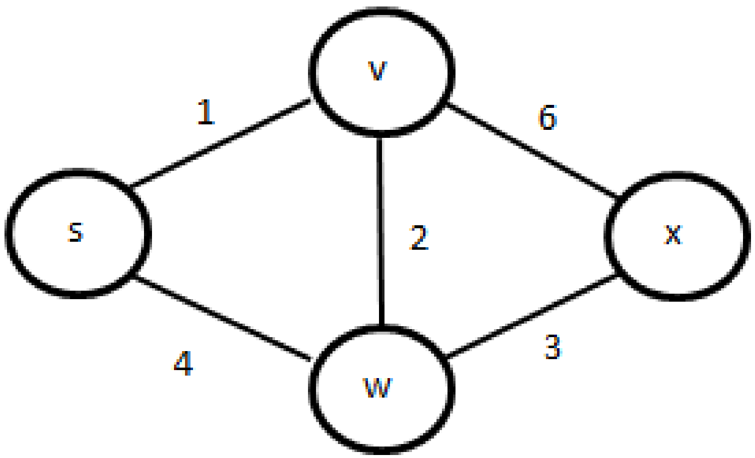
Aux Output: Array B with property that B[v] is a shortest path from s to v.

The Algorithm:

$A[s] \leftarrow 0$. $B[s] \leftarrow$ empty path (empty set)
$X \leftarrow \{s\}$ //Basis step

while $X \neq V$ **do**
{ POOL $\leftarrow \{(v,w) \in E \mid v \in X \text{ and } w \notin X\}$ }
$(v',w') \leftarrow$ search POOL for edge (v,w) for which A[v] + wt(v,w) is minimal
add w′ to X
$A[w'] \leftarrow A[v'] + wt(v',w')$
$B[w'] \leftarrow B[v'] \cup \{(v',w')\}$

Worked Example: Step 1



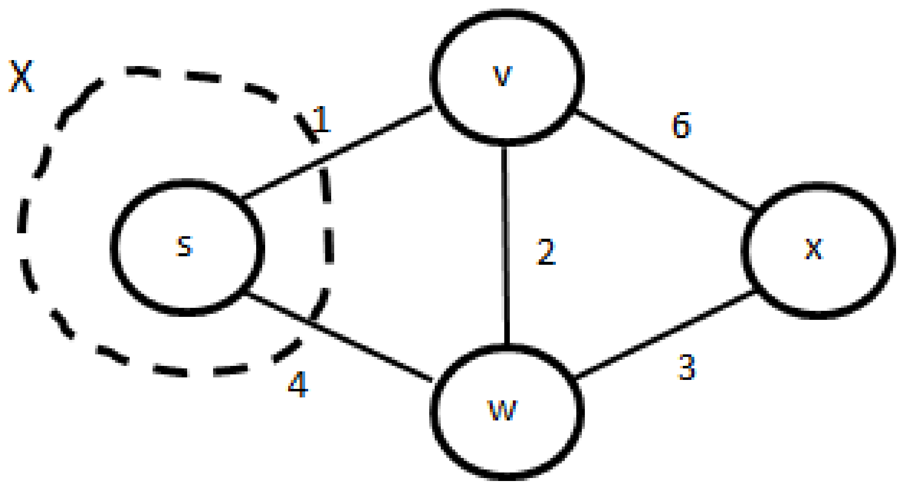
Step 1.

A[s] ← 0

B[s] ← { }

Put s in X

Worked Example: Step 2



Step 2.

X = { s }

POOL = { (s,v), (s,w) }

Find minimum greedy length – min of the following

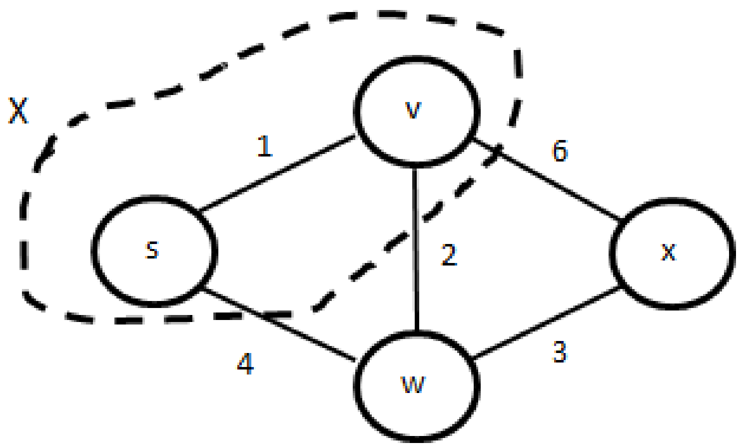
A[s] + wt(s,v) = wt(s,v) = 1 ←

A[s] + wt(s,w) = wt(s,w) = 4

Add v to X and set value of A[v]: A[v] ←1

Auxiliary Storage: B[v] = B[s] U {(s,v)} = {(s,v)}

Worked Example: Step 3



Step 3.

X = { s, v }

POOL = { (s,w), (v,w), (v,x) }

Find minimum greedy length – min of the following

A[s] + wt(s,w) = wt(s,w) = 4

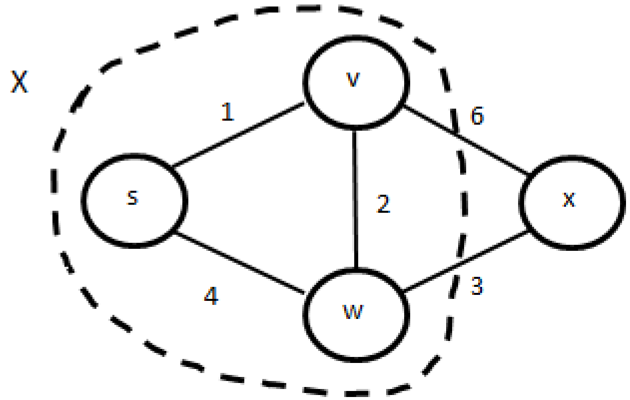
A[v] + wt(v,w) =1 + wt(v,w) = 1 + 2 = 3 ←

A[v] + wt(v,x) =1 + wt(v,x) = 1 + 6 = 7

Add w to X and set value of A[w]: A[w] ← 3

Auxiliary Storage: B[w] = B[v] U {(v,w)} = {(s,v), (v,w)}

Worked Example: Step 4



Step 4.

X = { s, v, w }

POOL = { (w,x), (v,x) }

Find minimum greedy length – min of the following

A[v] + wt(v,x) = 1 + wt(v,x) = 1 + 6 = 7

A[w] + wt(w,x) = 3 + wt(w,x) = 3 + 3 = 6 ←

Add x to X and set value of A[x]: A[x] ← 6

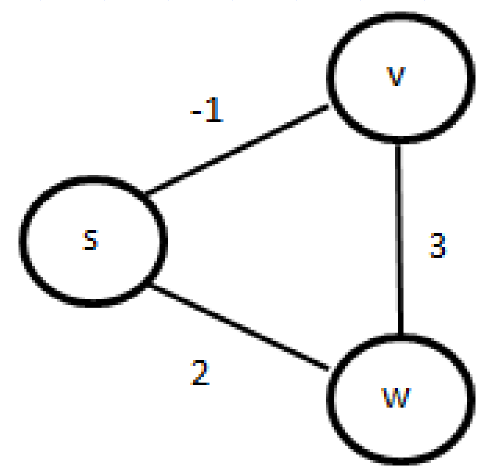
Algorithm complete since X = V. Computed values:

A[s] = 0   A[v] = 1   A[w] = 3   A[x] = 6

Computeed values of array B:
B[s] = { }, B[v] = {(s,v)}, B[w] = {(s,v), (v,w)}, B[x] = {(s,v), (v,w), (w,x)}

Dijkstra - Exercises

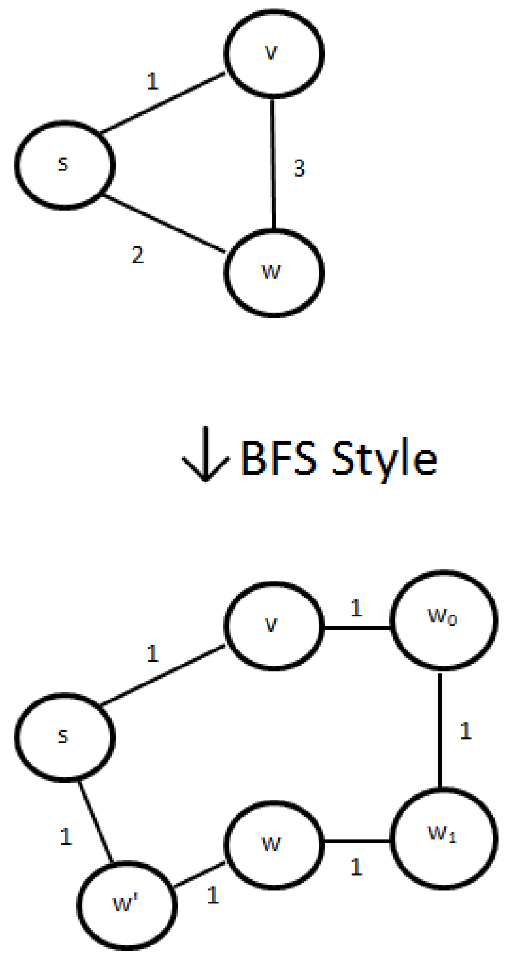
Why is there a requirement that edges have *non-negative* weights? Does Dijkstra’s Algorithm work correctly when there are negative edge weights? Consider this weighted graph.



Exercises, continued

Compare Dijkstra's approach to the shortest path problem with the approach of simply using BFS, as described in the previous lesson. Which one is better?

[BFS approach: Making all edge weights = 1 is same as removing all weights. Perform BFS with start vertex s and compute distance to each vertex by returning its *level* in the BFS spanning tree. These computed values should be same as values found using Dijkstra]



Dijkstra’s Algorithm As A Greedy Algorithm

- Dijkstra’s algorithm is an example of an *optimization* problem – trying to find an optimal solution among many possible solutions.
  - Optimization problem: Some quantity is to be minimized or maximized.
- Algorithms for optimization problems typically go through a sequence of steps, making choices along the way.
- If, in making such choices, an algorithm always chooses the option that appears best at the time, it is called a *greedy algorithm.*
- Dijkstra’s Algorithm is an example of this algorithm design strategy – select the next vertex to enter the cloud by choosing the one that yields least greedy length.

Optional: Correctness

- Loop Invariant: I(i) is the following statement: (where i means iteration #i)

  (1) $|X| = i + 1$

  (2) $A[v] = d(s,v)$ for all $v \in X$

Note: Please refer to the lecture notes to see why this algorithm works.

Dijkstra – Running Time

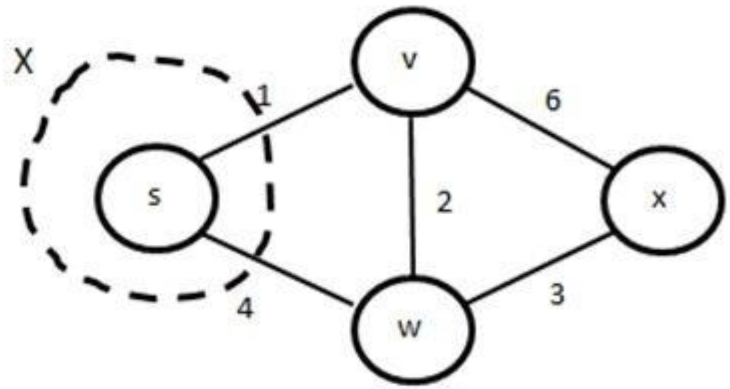
- Running time can be computed by observing that a (potentially) exhaustive search of edges is made in each iteration, leading to a running time of O(mn).
- This can be improved to O(m * log n) if an optimal data structure is used.

Improving Dijkstra

- Since “mins” are needed in each iteration, serve them using a Priority Queue instead of doing an exhaustive search of edges.
- Exhaustive search for next optimal vertex forces us to make *redundant computations* (next slides). These can be avoided by having the next optimal vertex *immediately available* at each step of the algorithm, and this can be accomplished by storing vertices in a Priority Queue in which vertices are prioritized according to optimal greedy lengths.

Redundant Computations in the Naïve Algorithm (1)

Steps 2 and 3 in worked example:



Step 2.

X = { s }

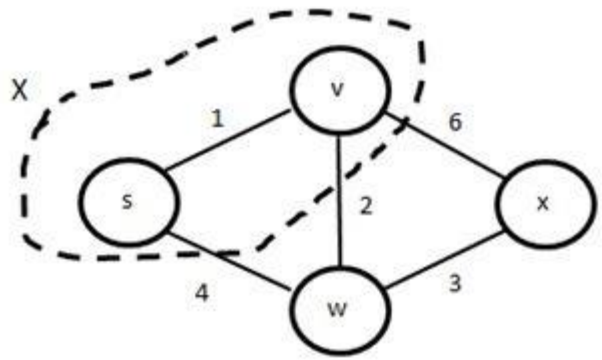
POOL = { (s,v), (s,w) }

Find minimum greedy length – min of the following

A[s] + wt(s,v) = wt(s,v) = 1

A[s] + wt(s,w) = wt(s,w) = 4 first computation

Add v to X and set value of A[v]: A[v] ←1



Step 3.

X = { s, v }

POOL = { (s,w), (v,w), (v,x) }

Find minimum greedy length – min of the following

A[s] + wt(s,w) = wt(s,w) = 4 second computation

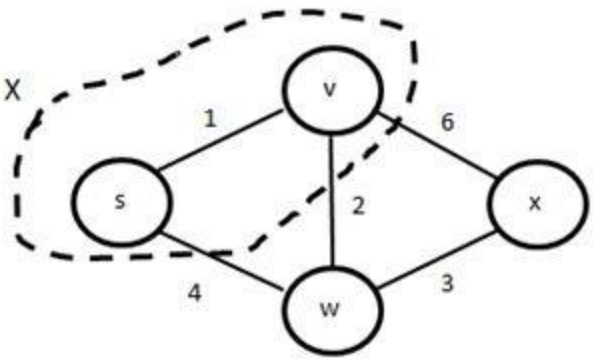
A[v] + wt(v,w) =1 + wt(v,w) = 1 + 2 = 3

A[v] + wt(v,x) =1 + wt(v,x) = 1 + 6 = 7

Add w to X and set value of A[w]: A[w] ← 3

Redundant Computations in the Naïve Algorithm (2)

Steps 3 and 4 in worked example:



Step 3.

X = { s, v }

POOL = { (s,w), (v,w), (v,x) }

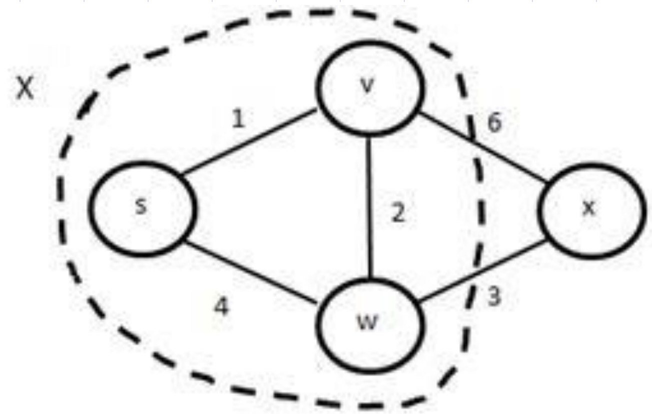
Find minimum greedy length – min of the following

A[s] + wt(s,w) = wt(s,w) = 4

A[v] + wt(v,w) = 1 + wt(v,w) = 1 + 2 = 3

A[v] + wt(v,x) = 1 + wt(v,x) = 1 + 6 = 7 first comput

Add w to X and set value of A[w]: A[w] ← 3



Step 4.

X = { s, v, w }

POOL = { (w, x), (v,x) }

Find minimum greedy length – min of the following

A[v] + wt(v,x) = 1 + wt(v,x) = 1 + 6 = 7 2nd comput

A[w] + wt(w,x) = 3 + wt(w,x) = 3 + 3 = 6

Add x to X and set value of A[x]: A[x] ← 6

Dijkstra – Using Priority Queue

The Algorithm:

```
A [s] <- 0
A[v] <- infinity (for each vertex v in V where v != s)
Q <- new heap-based priority queue  //items in Q are constructed by (u, A[u]) and ordered by A[u]
while !Q.isEmpty() do
    (u, A[u]) <- Q.removeMin()
    for each v adjacent to u and v is in Q :
        alt = A[u] + wt(u, v)
        if alt < A[v]
            A[v] <- alt
            Q.updateNode(v, alt)
return the label A[u] for each vertex u
```

Running time - updateNode

- updateNode(v, alt):
  - We can think it as two steps - delete node at v, and insert a new node (v, alt).
  - Insert a new node to priority queue takes O(log n)
  - Lab: Describe an algorithm for deleting a key from a heap-based priority queue that runs in O(log n) time.

Running time

- Initialize A[v] for all vertices. O(n)
- Build priority queue for all vertices O(n)
- The while loop removes one min node each time until the priority queue is empty. So the algorithm is going to execute while loop n times.
  - Remove min node and do downheap O(nlog n)
  - Computing the greedy length for each edge is done exactly once,
    if the greedy length computed at some particular point
    is less than what is stored on priority queue,
    we need to update this value on priority queue, O(mlog n)
- Since the graph is connected, n is O(m).
- Running time is therefore

  O(m log n)

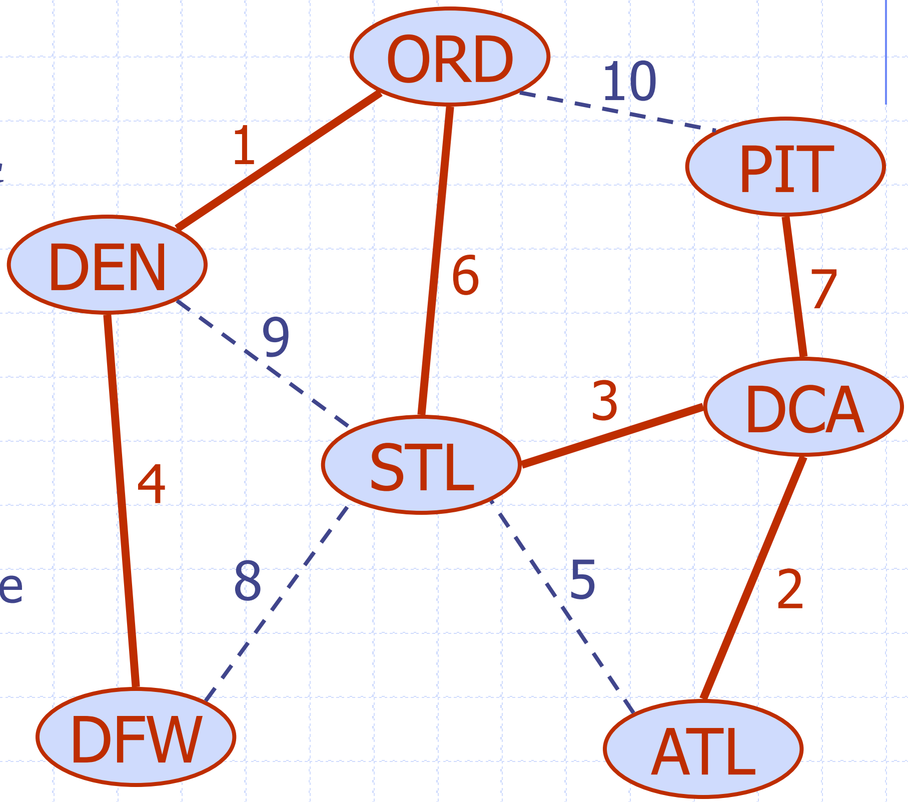
  This improves the O(mn) running time of the naïve algorithm.

Main Point

- Dijkstra's algorithm is an example of a *shortest-path algorithm* – an algorithm that efficiently (O(mlog n)) computes the shortest distance between two vertices in a graph.
- Analogously, Nature itself is known to obey the law of least action – Nature does the least possible amount of work to proceed from one location or state to another. Nature's way of achieving this makes use of computational dynamics that involve "no effort" and no steps.

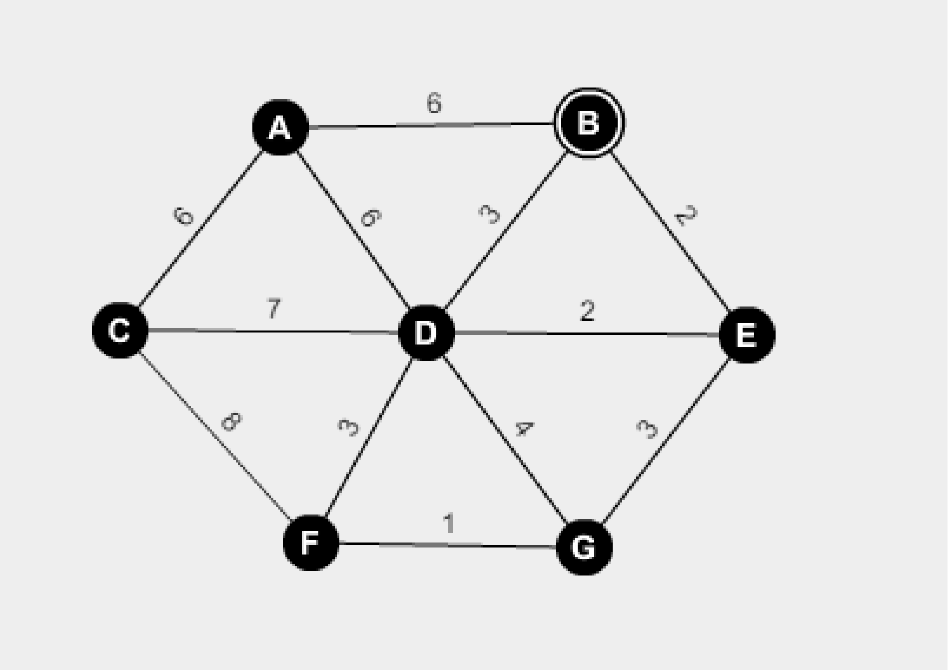
Minimum Spanning Tree

- Spanning subgraph
  - Subgraph of a graph G containing all the vertices of G
- Spanning tree
  - Spanning subgraph that is itself a tree
- Minimum spanning tree (MST)
  - Spanning tree of a weighted graph with minimum total edge weight

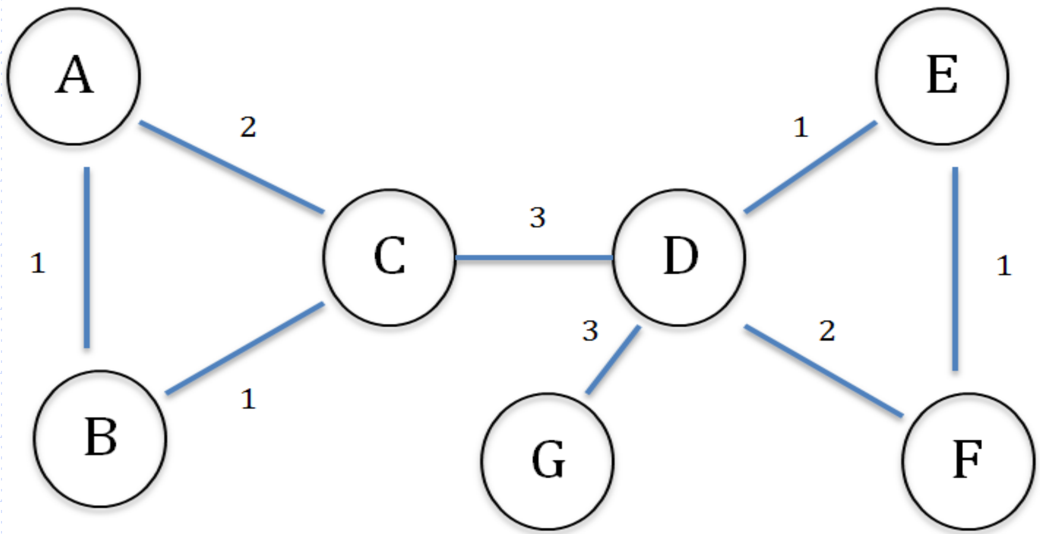


Application

CONNECTOR. The diagram below schematically represents potential railway paths between cities; a numeric label represents the cost to lay the track between the respective cities. What is the least costly way to build the railway network in this case, given that it must be possible to reach any city from any other city by rail? Devise an algorithm for solving such a problem in general.



Kruskal's Greedy Strategy



Build a collection T of edges by doing the following: At each step, add an edge e to T of least weight subject to the constraint that adding e to T does not create a cycle in T. To answer questions about correctness and running time of this algorithm, we need to specify certain details.

Implementation Questions

1. How do we pick the next edge at each step?

   Solution: We can arrange edges by sorting them by weight (in ascending order), and so we pick edges according to this sorted order.

2. How do we make sure that we do not add an edge to T that produces a cycle?

   Solution: We can ensure no cycles are created by building local minimum spanning trees around each vertex.

Kruskal's Algorithm

- First step is to sort all edges by weight.
- Second step involves creation of *clusters*
  - Every vertex is initially placed in a trivial *cluster* -- the cluster for a vertex v, denoted C(v), is simply {v}. A cluster represents a local minimum spanning tree.
  - When the next edge (u,v) is considered, C(u) and C(v) are compared -- if different, (u,v) is included as an edge in the final output tree, and C(u) and C(v) are merged.

Kruskal's Algorithm

Input: A simple connected weighted graph $G = (V, E)$ with n vertices and m edges
Output: A minimum spanning tree T of G

The Algorithm:

sort E in increasing order of edge weight

for each vertex v in G, define an elementary cluster $C(v)$ (which will grow)

by $C(v) = \{v\}$

$T \leftarrow$ an empty tree // T will eventually become the minimum spanning tree

while T has fewer than $n - 1$ edges **do**

$(u,v) \leftarrow$ next edge

$C(v) \leftarrow$ cluster containing v

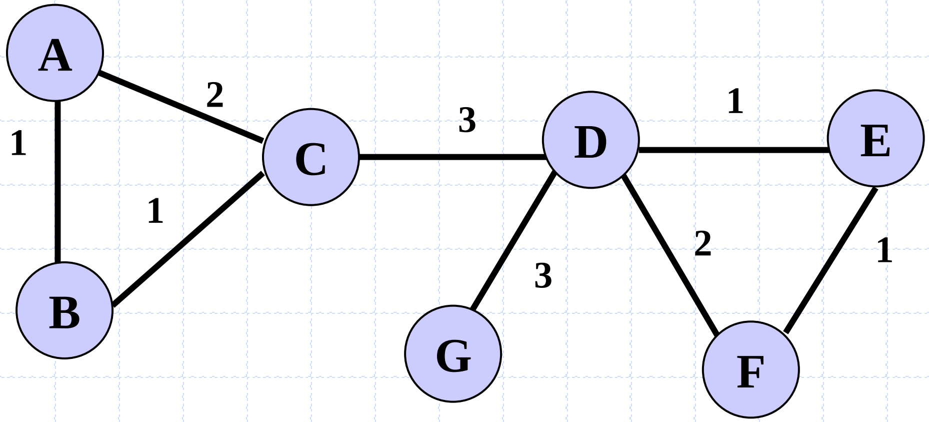
$C(u) \leftarrow$ cluster containing u

if $C(v) \neq C(u)$ **then**
add edge (u,v) to T
merge $C(u)$ and $C(v)$ (and update other clusters as needed)

return T

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG
T = {...}

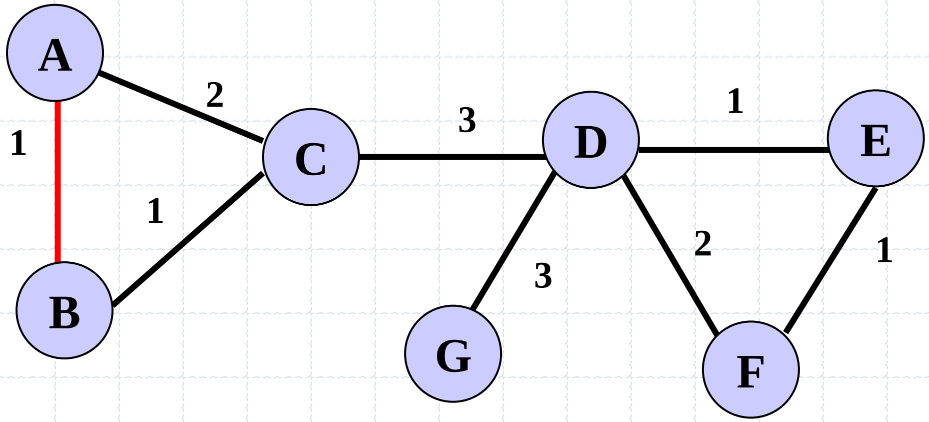


Cluster	*Evolving Values*
C(A)	{A}
C(B)	{B}
C(C)	{C}
C(D)	{D}
C(E)	{E}
C(F)	{F}
C(G)	{G}

Step 1:
Sort the edges and initialize the clusters

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG
T = {AB, …}

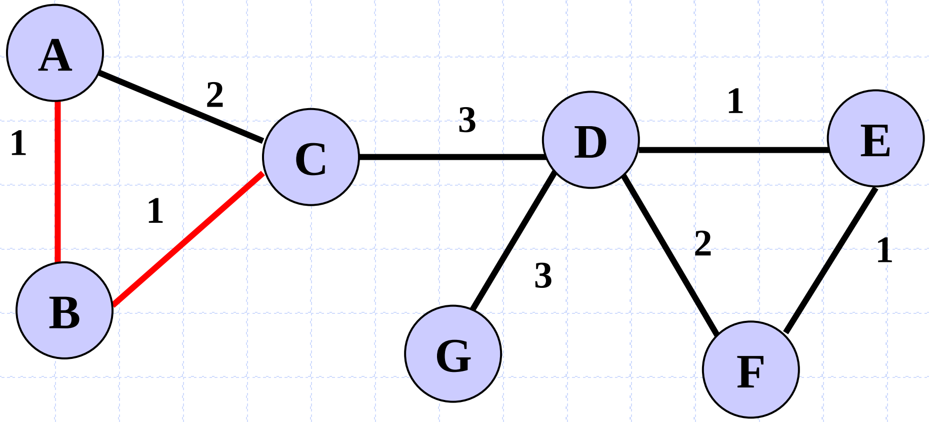


Cluster	*Evolving Values*
C(A)	{A, B}
C(B)	{A, B}
C(C)	{C}
C(D)	{D}
C(E)	{E}
C(F)	{F}
C(G)	{G}

Step 2:
$C(A) \neq C(B)$
add AB to T, merge C(A) and C(B)

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG
T = {AB, BC, ...}

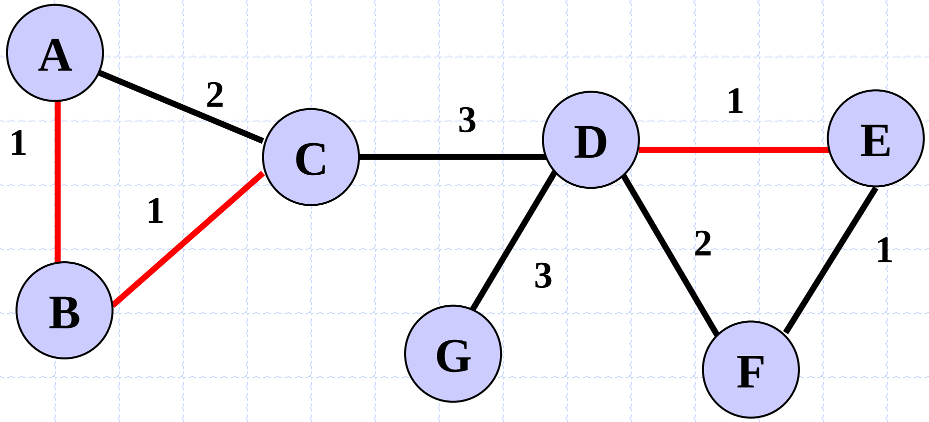


Cluster	*Evolving Values*
C(A)	{A, B, C}
C(B)	{A, B, C}
C(C)	{A, B, C}
C(D)	{D}
C(E)	{E}
C(F)	{F}
C(G)	{G}

Step 3:
C(B) ≠ C(C)
add BC to T, merge C(B) and C(C)

Worked Example

Sorted edges: <u>AB</u>, <u>BC</u>, <u>DE</u>, EF, AC, DF, CD, DG
T = {AB, BC, DE…}

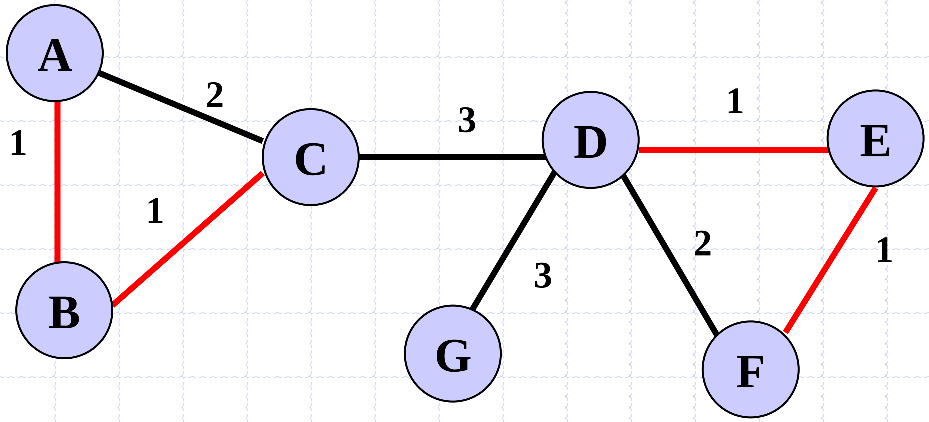


Cluster	*Evolving Values*
C(A)	{A, B, C}
C(B)	{A, B, C}
C(C)	{A, B, C}
C(D)	{D, E}
C(E)	{D, E}
C(F)	{F}
C(G)	{G}

Step 4:
C(D) ≠ C(E)
add DE to T, merge C(D) and C(E)

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG
T = {AB, BC, DE, EF, ...}

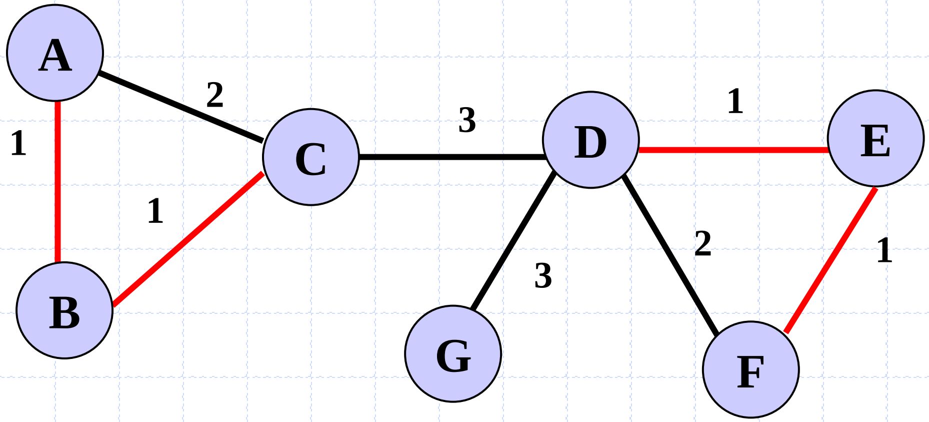


Cluster	*Evolving Values*
C(A)	{A, B, C}
C(B)	{A, B, C}
C(C)	{A, B, C}
C(D)	{D, E, F}
C(E)	{D, E, F}
C(F)	{D, E, F}
C(G)	{G}

Step 5:
C(E) ≠ C(F)
add EF to T, merge C(E) and C(F)

Worked Example

Sorted edges: <u>AB</u>, <u>BC</u>, <u>DE</u>, <u>EF</u>, <u>AC</u>, DF, CD, DG
T = {AB, BC, DE, EF, ...}

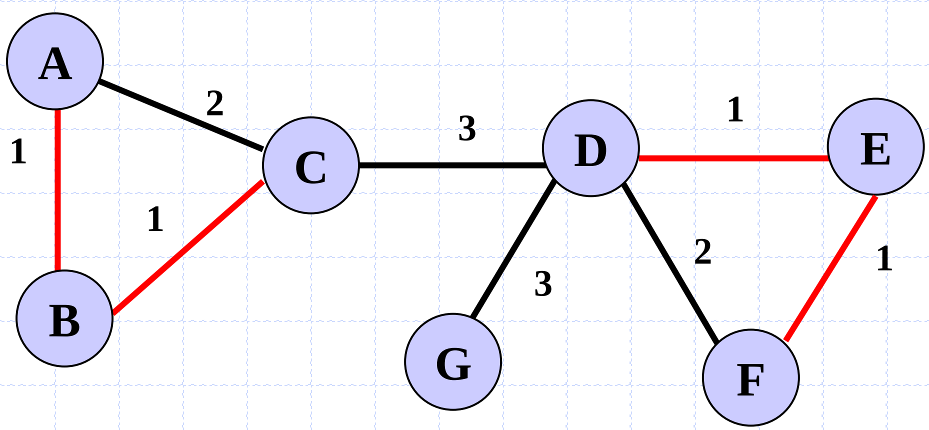


Step 6:
C(A) = C(C) , discard AC

Cluster	*Evolving Values*
C(A)	{A, B, C}
C(B)	{A, B, C}
C(C)	{A, B, C}
C(D)	{D, E, F}
C(E)	{D, E, F}
C(F)	{D, E, F}
C(G)	{G}

Worked Example

Sorted edges: <u>AB</u>, <u>BC</u>, <u>DE</u>, <u>EF</u>, <u>AC</u>, <u>DF</u>, CD, DG
T = {AB, BC, DE, EF, …}

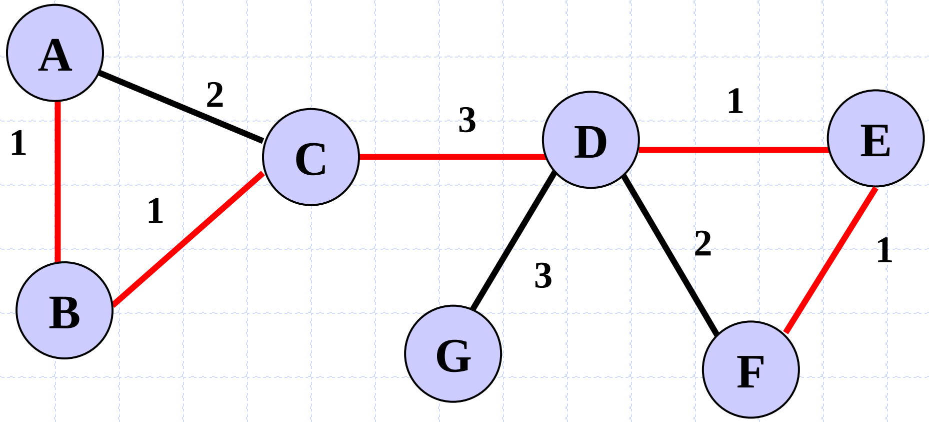


Step 7:
C(D) = C(F) , discard DF

Cluster	*Evolving Values*
C(A)	{A, B, C}
C(B)	{A, B, C}
C(C)	{A, B, C}
C(D)	{D, E, F}
C(E)	{D, E, F}
C(F)	{D, E, F}
C(G)	{G}

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG
T = {AB, BC, DE, EF, CD, ...}

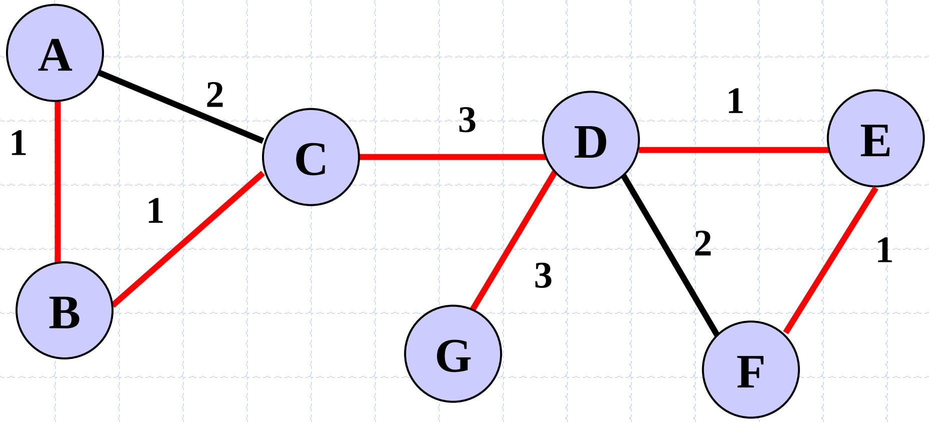


Cluster	*Evolving Values*
C(A)	{A, B, C, D, E, F}
C(B)	{A, B, C, D, E, F}
C(C)	{A, B, C, D, E, F}
C(D)	{A, B, C, D, E, F}
C(E)	{A, B, C, D, E, F}
C(F)	{A, B, C, D, E, F}
C(G)	{G}

Step 7:
C(C) ≠ C(D)
add CD to T, merge C(C) and C(D)

Worked Example

Sorted edges: AB, BC, DE, EF, AC, DF, CD, DG
T = {AB, BC, DE, EF, CD, DG, …}

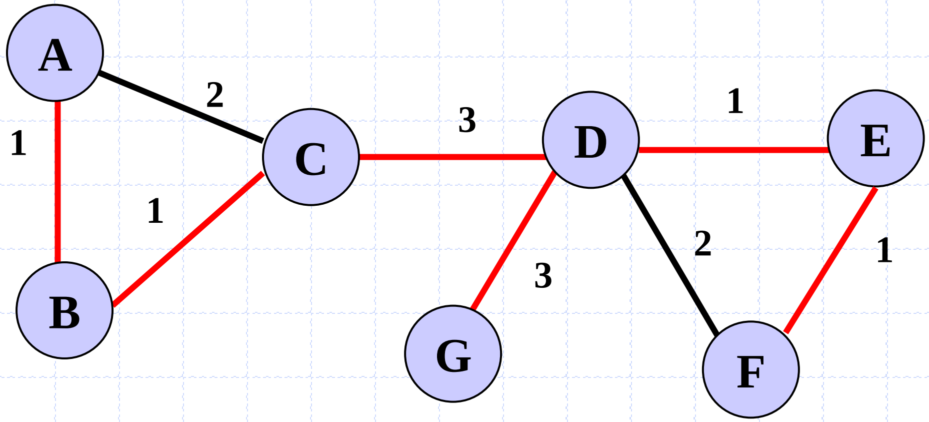


Step 8:
C(D) ≠ C(G)
add DG to T, merge C(D) and C(G)

Cluster	*Evolving Values*
C(A)	{A, B, C, D, E, F, G}
C(B)	{A, B, C, D, E, F, G}
C(C)	{A, B, C, D, E, F, G}
C(D)	{A, B, C, D, E, F, G}
C(E)	{A, B, C, D, E, F, G}
C(F)	{A, B, C, D, E, F, G}
C(G)	{A, B, C, D, E, F, G}

Worked Example

Sorted edges: <u>AB</u>, <u>BC</u>, <u>DE</u>, <u>EF</u>, <u>AC</u>, <u>DF</u>, <u>CD</u>, <u>DG</u>
T = {AB, BC, DE, EF, CD, DG}



Cluster	*Evolving Values*
C(A)	{A, B, C, D, E, F, G}
C(B)	{A, B, C, D, E, F, G}
C(C)	{A, B, C, D, E, F, G}
C(D)	{A, B, C, D, E, F, G}
C(E)	{A, B, C, D, E, F, G}
C(F)	{A, B, C, D, E, F, G}
C(G)	{A, B, C, D, E, F, G}

Now we have n-1 = 6 edges in T, the algorithm stops.

Optional: Correctness

We need to verify the following Facts:

1. During execution, distinct clusters are always disjoint, and for each cluster C, if T[C] is the subgraph of G whose set of vertices is C and whose edges are those edges of T whose endpoints lie in C, then T[C] is a tree.
2. No cycle ever arises in T during execution of the algorithm
3. The main loop terminates (it is conceivable that after all edges have been examined, T still contains $< n - 1$ edges – this is shown to be impossible)
4. The set T that is returned is a spanning tree for G
5. The set T is a *minimum* spanning tree for G.

Note: Please refer to the lecture notes to see why this algorithm works.

Running Time of Kruskal: First Try

- Time to sort edges: $O(m\log n)$
- Cost of while loop = $O(mn)$
  - loop potentially accesses every edge
  - comparison $C(x) = C(y)$, with a hashtable implementation of sets, is $O(1)$
  - merging $C(x)$, $C(y)$ costs $\min\{|C(x)|, |C(y)|\}$, which is $O(n)$.
- Cost of while loop can be improved with a good choice of data structure

DisjointSets Data Structure

- Data structure for maintaining a partition of a set into disjoint subsets (data structure sometimes called *Partition* rather than DisjointSets)
- General features
  - *Data:*
    - Universe U – the base set that is being partitioned (this set is never altered)
    - Collection $\mathcal{C} = \{X_1, X_2, \ldots, X_n\}$ of subsets of the universe – the subsets are disjoint and their union is U (these subsets are modified when the data structure is used – size of $\mathcal{C}$ shrinks because of repeated union operations)
  - *Operations:*
    - find(x) – returns the subset X_i to which x belongs
    - union(A,B) – replaces the subsets A, B in C with A U B.

Example

- Initial Structure:
  - $U = \{1, 2, 3, 4, 5\}$
  - $X_1 = \{1, 2\}, X_2 = \{3\}, X_3 = \{4, 5\}$
  - $C = \{X_1, X_2, X_3\}$
- find Operation:
  $find(2) = X_1$ $find(5) = X_3$
- union Operation:
  $union(X_1, X_2) = X_1 \cup X_2 = \{1,2,3\}$
  New value for C is $\{\{1,2,3\}, \{4,5\}\}$

Tree-Based Implementation of DisjointSets

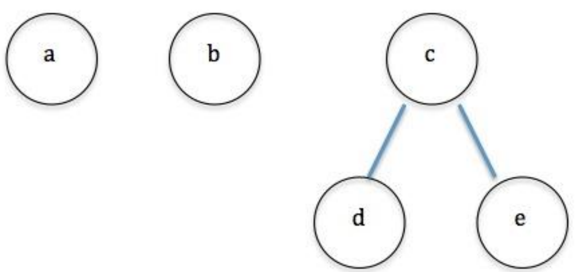
- The elements of each set X in the collection *C* are represented by nodes in a tree T_X; the set X itself is referenced by its root r_X.
- find(x) returns the root of the tree to which x belongs
- union(x,y) joins the tree that x belongs to to the tree that y belongs to by pointing root of one to the root of the other.

Example

U = {'a', 'b', 'c', 'd', 'e'}

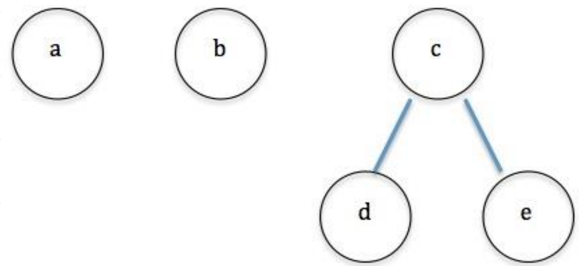
C = {{'a'}, {'b'}, {'c', 'd', 'e'}}

Tree representations:

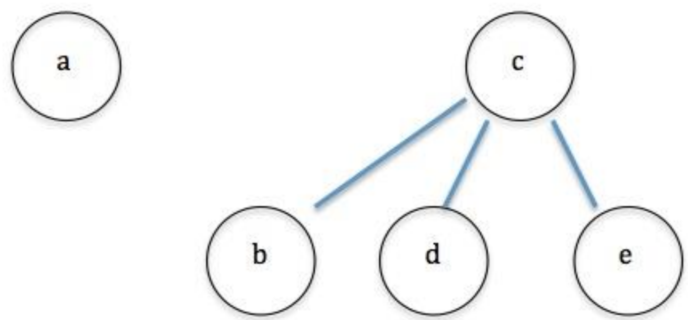


- find('d') returns 'c'

Example (cont)



- union('b', 'c') points root 'b' to root 'c'



Now, find('b') returns 'c'

Code

```
//handle trees by keeping track of parents only
//whenever a character c is a root, its parent is set to be c itself
HashMap<Character, Character> parents = new HashMap<Character, Character>();
char[] universe;

//find returns the root of tree representing a subset
//worst case: find requires full depth of tree to locate root of representing tree
public char find(char element) {
    char nextParent = parents.get(element);
    if(nextParent == element) {
        return element;
    } else {
        return find(nextParent);
    }
}
```

Code

```
//union() accepts only tree roots (representing subsets) as arguments
//The method simply points the first root to the second
//In the worst case, resulting tree is taller than original two
public void union(char a_tree, char b_tree) {
    parents.put(a_tree, b_tree);
}
```

To avoid building up trees that are too tall (and therefore imbalanced), an optimization can be used: Always point the shorter tree's root to that of the taller.

Optimized Code

```
HashMap<Character, Character> parents = new HashMap<Character, Character>();
char[] universe;
//keep track of heights of trees
HashMap<Character, Integer> heights = new HashMap<Character, Integer>();

public void union(char a_tree, char b_tree) {
    int height_a = heights.get(a_tree);
    int height_b = heights.get(b_tree);
    if(height_a < height_b) {
        parents.put(a_tree, b_tree);
    }  else if(height_b < height_a) {
        parents.put(b_tree, a_tree);
    }  else { //height_a == height_b
        parents.put(a_tree, b_tree);
        heights.put(b_tree, height_b + 1); //this is case in which height is increased
    }
}
```

See https://en.wikipedia.org/wiki/Disjoint-set_data_structure

Optimized Running Time of Kruskal

- Time to sort edges: O(mlog n)
- Cost of while loop = O(mlog n)
  - loop potentially accesses every edge //O(m)
  - comparison C(x) = C(y) follows these steps:
    //locates roots of representing trees
    - $r_x \leftarrow$ find(x) and $r_y \leftarrow$ find(y) //O(log n)
    - check whether $r_x = r_y$ //O(1)
  - merging C(x), C(y) is done by union() operation //O(1)

- Total (optimized) running time for Kruskal: O(mlog n).

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. A Minimum Spanning Tree can be obtained from a weighted graph G = (V,E) by examining all possible subgraphs of G, and extracting from those that are trees having the smallest sum of edge weights. This procedure runs in $\Omega(2^m)$, where n = |E|.
2. Kruskal's Algorithm is a highly efficient procedure (O(mlog n)) for finding an MST in a graph G. It proceeds by choosing edges with minimum possible weight subject to the constraint that selected edges do not introduce a cycle in the set T of edges obtained so far.

3. *Transcendental Consciousness*, the simplest form of awareness, is the source of effortless right action.
4. *Impulses Within the Transcendental Field.* Effortless, economical, mistake-free creation arises from the self-referral dynamics of the field of pure consciousness.
5. *Wholeness Moving Within Itself.* In Unity Consciousness, optimal solutions arise as an effortless unfoldment within one's unbounded nature.